Memory Management in the Tera MTA Computer System

Authors:
Richard Korry
Tera Computer Company
2815 Eastlake Avenue East
Seattle, Washington 98102
richk@tera.com
(206)325-0800
Cathy McCann
mccann@tera.com
Burton Smith
burton@tera.com

Keywords:
memory scheduling, swapping, batch scheduling, space-sharing

Abstract

This paper describes memory scheduling for the Tera MTA (Multi Threaded Architecture) computer system. The Tera MTA is intended to support a mixture of large and small tasks running in parallel, and ensure that they all make progress commensurate with their importance. We describe the memory scheduling algorithms used to schedule these tasks fairly. Some of the issues encountered and solutions proposed are novel, due in part to the highly multiprogrammed nature of our architecture. In particular, we present an algorithm for swapping a set of tasks to and from memory that achieves minimal overhead, largely independent of the order in which tasks are swapped.

1. Introduction

The Tera MTA implements a multi-user system in which many parallel jobs may execute concurrently. Each multi-threaded processor can support up to 16 jobs simultaneously that dynamically compete for the processor's instruction streams. Memory must be scheduled for these jobs because there are competing jobs executing in parallel and because of the broad spectrum of job memory requirements.

The workload anticipated for the MTA is varied. We expect most sites to have large, parallel jobs intermixed with interactive work. The operating system scheduler must excel in this environment. Large parallel jobs require high throughput, while short, interactive jobs require quick response. Also, the scheduler must prevent starvation and ensure some measure of fairness among jobs of similar characteristics; to facilitate this, we provide a mechanism for job differentiation.

This paper describes the memory scheduling issues associated with a multipurpose high performance parallel computer system like the Tera MTA. A brief discussion of the architecture is followed by a section on the operating system that describes the overall goals of job memory scheduling. The next section describes in detail the algorithms used for memory scheduling. We end with a few conclusions

and a list of references.

2. Architecture

The Tera MTA architecture is described in *Alverson et al.*. It implements a true shared-memory multiprocessor with multithreaded computational processors (CPs) and interleaved memory units connected by a packet-switched interconnection network. All memory units are addressable by each processor. There are no data caches; memory latency is tolerated through a combination of multithreading and register prefetching. Programmable input/output processors (IOPs) transfer data to and from the memory and the attached peripheral devices. The network has enough bisection bandwidth to support one request and one response from each processor per cycle.

A CP supports 16 protection domains, each of which has a memory map. Multiple hardware threads can be executing in each protection domain of a CP simultaneously. One domain is used for operating system daemons, so the remaining 15 protection domains let a processor execute as many as 15 distinct parallel user applications at once. The address space is segmented, with up to 16K segments per protection domain in sizes varying from 8 Kbytes to 256 Mbytes. Contiguous 256 Mbyte segments generate a contiguous, shared virtual address space of up to 4 Tbytes. The current implementation of the architecture supports 512 Gbytes of physical memory. It was the need for a large virtual address space that led us to choose segmentation rather than paging for address mapping.

An IOP deals with one input segment and one output segment at a time. An IOP instruction is used to switch segments. Peripheral devices are connected to IOPs via duplex 64-bit HIPPI channels capable of 200 Mbyte/s in both directions at once. The disk arrays (RAIDs) used for the file system and backing store can achieve over 150 Mbyte/s. IOPs not attached to peripherals are used in *loopback mode* to copy data from one part of memory to another at 1.3 Gbyte/s. Good copying bandwidth helps deal with the external fragmentation that non-paged segmentation gives rise to.

3. Operating System

The operating system allocates the resources of the system among tasks competing for these resources. The memory scheduler determines which subset of the ready tasks to load into memory. The processor scheduler then determines the subset of in-memory tasks to load onto the available protection domains. This document deals with the memory scheduler; the processor scheduler is the subject of a companion paper.

Multi-threaded tasks running on a shared memory multiprocessor like the Tera often communicate and synchronize frequently. If a thread faults and waits for the virtual memory system to load an unmapped region of memory, the remaining threads are likely to block soon thereafter, waiting to synchronize with the faulting thread. Thus demand-mapped virtual memory can generate performance problems on multiprocessors - see *Burger et al.* We avoid these problems by swapping entire tasks, running a task only when all of its segments are loaded.

The operating system memory scheduler distinguishes between tasks with large and small memory usage. Typically, large memory tasks execute for a long time, can tolerate being swapped out for a long time, and are usually submitted via a batch system such as the Network Queueing System (NQS). Swapping large memory tasks must be infrequent because of the high overhead per swap. When such a

task *is* interactive, its presence in memory must be explicitly arranged for.

Small-memory tasks have the inverse characteristics: they are short-lived or use resources in bursts (are interactive), do not tolerate being swapped out for long, and are usually submitted from a command shell. Small memory tasks may be swapped in and out of memory more frequently. However, for interactive computing, the user expects an immediate response to a keystroke. These tasks require frequent access to the processors to enjoy good interactive response.

Because large-memory and small-memory tasks differ in their memory usage patterns, we introduce mechanisms for scheduling memory that differentiate these two classes of workload. Studies of interactive and batch workloads by *Ashok and Zahorjan* support this differentiation. The memory resources are thus divided between large, batch oriented tasks and short, interactive tasks. Two memory schedulers are employed, the MB-scheduler for large (big) memory tasks and the ML-scheduler for small memory (little) tasks. Data memory is statically partitioned at boot time into two parts; one part for each scheduler. There is one MB-scheduler and one ML-scheduler per machine.

A new task is scheduled by the MB-scheduler if its memory requirement exceeds a site-defined value, otherwise it is scheduled by the ML-scheduler. When a task is swapped in, it is assigned to a processor scheduler. If a task scheduled by the ML-scheduler requests more memory than a site-specific threshold, it is handed over to the MB-scheduler when it is next swapped out. A task scheduled by the MB-scheduler whose memory requirements drop below the threshold remains under the control of the MB-scheduler. This avoids thrashing between the two memory schedulers.

4. Memory Scheduling

Despite its 1-2 GB of memory per processor, large applications will make memory a scarce resource (possibly *the* scarce resource) on the machine. Therefore, good performance requires efficient memory utilization. Starvation avoidance is also important, especially for tasks with large memory requirements.

As discussed in the previous section, memory is controlled by two schedulers: the MB-scheduler for tasks with large memory requirements and the ML-scheduler for small interactive tasks with small memory requirements. We assume large memory tasks have long execution times and can tolerate being swapped out for long periods of time. We assume small memory tasks are interactive, require frequent access to processor resources, and therefore cannot be swapped out for extended periods. Each site designates what portion of memory is controlled by which scheduler. Generally, most memory will be given to the MB-scheduler. Both schedulers execute the same algorithm to schedule the memory within their domains.

The memory scheduling algorithm in general must divide available memory among the set of tasks over time. Thus the scheduling space can be represented in two dimensions, with memory on the horizontal axis and time on the vertical axis. A *schedule* consists of a partitioning of memory over time among a subset of ready tasks in the system. An *allocation* to task j assigns some $m(j)$ units of memory for some period of time. A valid schedule ensures that the total memory allocated at any moment does not exceed system capacity. The goals of the memory scheduler are to minimize the total unallocated memory over time and to ensure high task throughput.

Both memory schedulers maintain a *ranking* of tasks within their domains. A ranking or priority of tasks is commonly used in schedulers to provide a way for users to specify the relative importance of task

resource allocation decisions and to avoid starvation by changing a task's ranking as it acquires resources. The memory schedulers use the ranking to determine the order in which tasks are considered for scheduling. The manner in which task ranking is defined is discussed below.

A memory schedule defines a set of tasks that will be resident at any one time. For each task, the scheduler determines when the task is scheduled to be in memory and for how long. The time during which a task is in memory is called its *dwell time*. The length of time tasks are in memory is dependent in part upon the overhead cost of swapping the task.

As discussed previously, the operating system requires that the entire address space of a task be resident while the task is executing. Thus, our model of I/O assumes that no task can start executing until its entire address space is swapped in, and no tasks can continue to make progress once any of its address space is swapped out.

Before describing the memory scheduling policy, we define a model for swapping overhead applicable to the MTA and many other parallel systems and use this model to define the cost of swapping overhead. We show that based on the overhead of swapping, it is appropriate for a task's dwell time to be set in proportion to the amount of memory the task requires.

Swapping Overhead

Periodically, the memory scheduler will reassign a subset of memory to a different set of tasks. The cost of memory scheduling is determined by the cost of swapping out a set of tasks and swapping in another set of tasks at these memory quantum intervals.

We use a simple model of I/O that defines a constant available swapping bandwidth r, and computes the time to swap m memory units as m / r. Thus, we assume large memory tasks utilize all the I/O bandwidth that is available for swapping if there are no other swapping activities in progress simultaneously. We define the overhead of swapping out a set of tasks and swapping in another set of tasks as the total space-time (e.g., byte-seconds) that memory is not available for execution.

We define the following swapping algorithm for swapping out a set of tasks and swapping in another set of tasks.

1. Choose *inTask*, the first task to be swapped in, and *outTask*, the first task to be swapped out, arbitrarily. Stop *outTask* by notifying its processor scheduler.
2. Begin swapping *outTask* out, and continue until either
   - there is enough room for *inTask* or
   - *outTask* is fully swapped out.
3. Swap all or part of *inTask* in to fill the memory vacated by *outTask*.
4. If *inTask* is now fully swapped in, notify its processor scheduler that it is ready to begin execution, and choose another *inTask*, if there is one.
5. If *outTask* was fully swapped out, choose another *outTask*, if there is one, and stop it.
6. While there is work remaining, return to step 2.

We alternate in this fashion between *inTask* and *outTask* to:

- start each *inTask* as early as possible, and

- stop each *outTask* as late as possible.

The overhead of the swapping algorithm is depicted in Figure 1a illustrates the overhead of swapping when these tasks are chosen for swapping (in or out) in ascending order of size. The total space-time overhead required to start a task consists of the time to swap out the task or tasks currently allocated the memory and the time to swap in the new task.

For example, to swap out task **d1** of size m requires m / r time during which m units of memory is unavailable. Each swap (in or out) of size m consumes an overhead depicted as a lightly-shaded rectangle of width m and height m / r for a total memory-time overhead area of $m*m / r$.

To swap in task **a1** of size m requires additional overhead of $m*m / r$ for a total overhead cost of $2*m*m / r$ to swap out one task for another of the same size. The total overhead cost of swapping is the summation of the overheads of the individual swapping events.

Figure 1b illustrates the swapping overhead for a different ordering of tasks swapped in. Each partial swap-out is accompanied by a partial swap-in of the same size. The lightly shaded rectangles in the figure depict the overhead of these swapping events. As before, the combined time space-time for a partial swap-out/swap-in pair of an amount of memory m is $2*m*m / r$, but total cost of this overhead is different from that in Figure 1a since the values of m reflect a partial swap-in or swap-out of a task.

The partial swap-out/swap-in pairs occur sequentially. The northeast and southwest vertices of the swap-in/swap-out rectangles form a line with a slope of $2 / r$. The dark-shaded area above this line represents the time during which a portion of unutilized memory is allocated to an incoming task. This memory is not utilized because the entire task is not memory resident. Similarly, the dark-shaded area below the line represents the time during which a portion of unutilized memory is allocated to an outgoing task; the memory is not utilized because the task is suspended while it is being swapped out.

The arrows indicate times at which a task is entirely swapped in (and can start execution) or at which a task to be swapped out stops executing.

The total overhead of swapping is the summation of the space-time overhead above and below the line. This space pictorially forms a staircase shape, where the width of each step is the size of memory for the incoming or outgoing task. The height of each step can be computed from the slope of the line.

If we let I be the set of tasks to be swapped in and D be the set of tasks to be swapped out. The total area representing the overhead above the line is the sum over all i in I of $m(i)*m(i) / r$, where $m(i)$ is the memory size of task i. Similarly, the area below the sloped line is the sum over all d in D of $m(d)*m(d) / r$. Thus, the total overhead incurred is the sum over all k in $I \cup D$ of $m(k)*m(k) / r$.

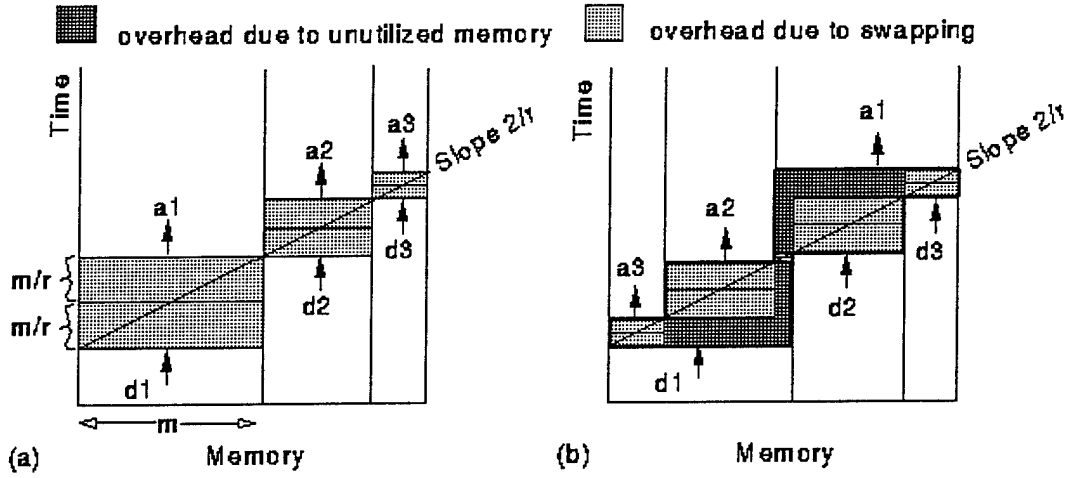

Figure 1. Swapping Overhead

It is easy to see that this algorithm achieves the minimal swapping overhead. Certainly, there is no benefit in discontinuing swapping out one task in order to swap out another, or discontinuing swapping in one task in order to swap in another; these actions only increase the overhead. Also, no benefit is gained by swapping multiple tasks simultaneously. (In fact, although our simple I/O model does not reflect it, additional overhead would be incurred from contention between two simultaneous swapping events.)

Our policy, which completes the swap-in of one task before starting the swap-in of another and completes the swap-out of one task before suspending another task to be swapped out minimizes the time during which memory is not utilized. Furthermore, our swapping algorithm yields the same overhead for any sequential ordering of jobs to be swapped out and any ordering of jobs to be swapped in.

In summary, the swapping is simplified since the overhead of swapping is independent of the order with which tasks are swapped in or out. Memory scheduling is also simplified since the overhead of swapping out tasks whose dwell times expire concurrently is the same as the cumulative overhead of a schedule that staggers the expiration of dwell times. Furthermore, a task's contribution to the swapping overhead grows quadratically with its size. To maintain a constant percentage overhead per task, the dwell time of a task should be proportional to its size.

Finally, our I/O model assumes that an entire task's address space must be resident in memory before a task can execute. While other systems may provide a memory replacement policy that allows tasks to continue executing with a partially resident working set (in spite of the performance problems that result), we believe the working set for parallel applications comprises a significant percentage of the total pages, implying the swapping overhead for demand-paged systems is similar.

User Demand

Typically, a task's importance is represented by the priority level it is assigned. Tasks at higher priority

levels receive preferential consideration in the allocation of resources. In a pure priority system, high priority tasks can starve tasks of lower priority by consuming all the resources. When feedback mechanisms are introduced to prevent starvation by adjusting priority upward when a task doesn't run and downward when it does, relative task importance is muddied. The result is an imprecise quantitative relationship between priority and average rate of resource consumption.

On the Tera MTA, a task's importance is expressed instead in terms of its nominal average resource allotment. The user defines a demand of memory resource consumption for a task, in units of space-time memory residency per unit time. This demand is the time-averaged nominal memory usage of the task. Presumably, higher demands justify increased monetary charges. If the system is saturated or underutilized, the demand will overestimate or underestimate, respectively, the average memory occupied by the task. The purpose of the demand parameter is to permit a quantifiable differentiation among tasks that is tied to system performance. The memory scheduler uses this demand to determine the order in which tasks are considered for execution.

Tasks are ordered by rank. The memory scheduler defines a task's rank as a linear function of the ratio between its demand of memory consumption and its achieved consumption. A task's rank is re-evaluated as it accumulates time in memory. Let *demand(i)* be the demand for task *i*. Then the rank of task *i* at a time *t* is defined as the sum of two terms:

$$rank(i,t) = timeAtCross(i,t) + (dwellTime(i) * memory(i)) / demand(i)$$

Here *dwellTime(i)* is the dwell time for task *i*, *memory(i)* is the size of task *i*, and *timeAtCross(i,t)* for task *i* is the time at which the memory resource consumption acquired so far equals its demand:

$$timeAtCross(i,t) = start(i,t) + totalConsumption(i,t) / demand(i)$$

where *totalConsumption(i,t)* is the total memory consumption acquired by task *i* up to time *t*. When a task arrives, its *timeAtCross* is set to an initial value *start(i,t)*; when a task swaps out, the memory resource just consumed is divided by the task's *demand* is added. This scheme is philosophically similar to *fair share* (*Henry*, *Essick*) and *highest penalty ratio next* (*Finkel*) scheduling. Unlike those methods, however, *rank(i,t)* only needs to be updated when a task is swapped out.

Note that *timeAtCross* can be either in the past or in the future, depending on system load. The initial value of *timeAtCross* can therefore be used to regulate when new tasks begin to contend for memory. If *start(i,t)* is *t*, the current time, a lightly loaded system will swap in a new task immediately whereas a heavily loaded system will defer it until the older tasks have caught up. If *start(i,t)* is *timeAtCross* of the most recently scheduled task, the new task will immediately begin to compete with the older tasks. Finally, if we let *start(i,t)* be its smallest possible value, the new task will swap in and stay in.

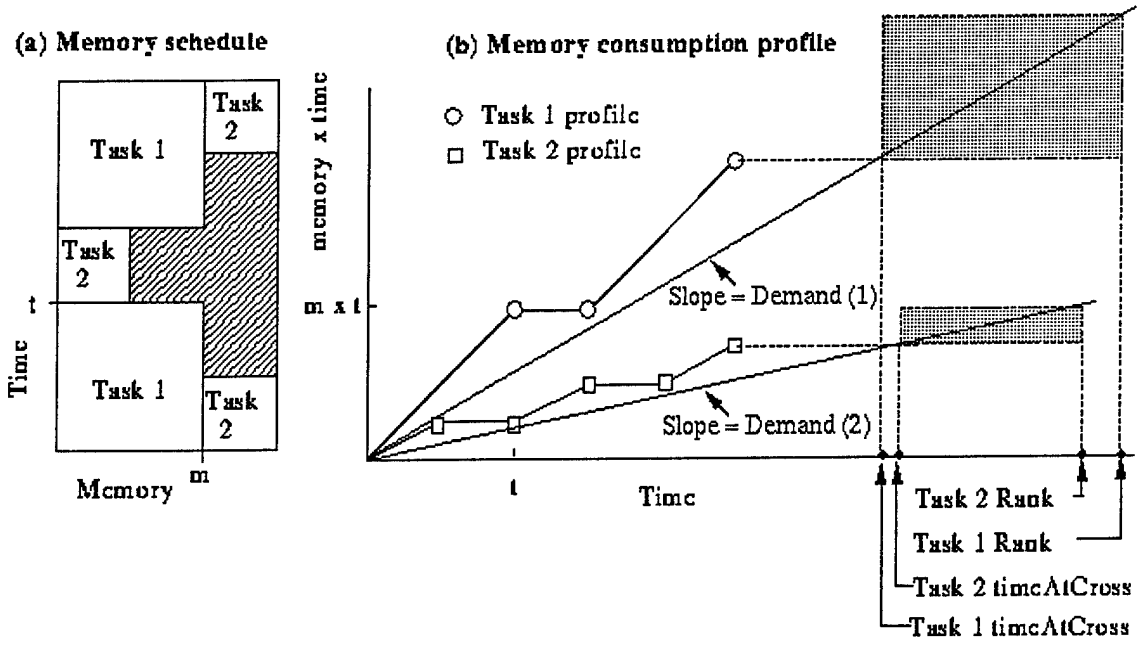

Figure 2. Task Ranking Strategy

To illustrate this ranking, Figure 2a shows an example memory schedule for two tasks with the same start(i,t). The shaded areas represent allocations to other tasks not of interest in this example. Figure 2b shows the memory resource consumption profile for tasks 1 and 2 as a result of the schedule. For example, after Task 1 executes for time *t*, it has acquired *m* * *t* memory resources where *m* is its size. Task 2 has acquired *m* * *t* / *4* of memory-time by the same time, *t*.

The shaded rectangles in Figure 2b illustrate pictorially the computation of rank for Tasks 1 and 2 after the execution of the memory schedule. A task's demand is represented by a sloped line. Conceptually, the first term of rank, *timeAtCross*, attempts to order tasks according to their urgency in acquiring memory resources to meet their desired rate. This is determined by a horizontal line from the tasks memory consumption profile to its demand slope. Tasks with a earlier *timeAtCross* are considered more crucial.

The second term of rank compensates for the different rate at which tasks acquire resources during a quantum (*dwellTime*) and their different resource demands. The shaded rectangles have height *dwellTime(i)* * *memory(i)*; because their diagonals have slope *demand(i)*, their width is *dwellTime(i)* * *memory(i)* / *demand(i)*. Given two tasks with the same *timeAtCross*, the task with the higher ratio of per-quantum memory consumption to demand will be considered less crucial. It can wait longer before being scheduled and still achieve its demand.

It is easy to see that *rank(i,t)* can be interpreted as the value *timeAtCross(i,t)* would assume if it were given another quantum. From this example, after the current schedule is complete, the rank for task 2 is less than the rank for task 1, indicating that task 2 will be considered for scheduling before task 1.

Memory Scheduling Policy

The MB-scheduler and the ML-scheduler use the same algorithm for allocating memory to a set of tasks. Each scheduler decides which tasks to move in or out of memory based on a ranking of the tasks in the system, where the ranking reflects the ratio of the task's demand and the achieved rate of memory consumption. The goal of the memory scheduler is primarily to provide high memory utilization while achieving the overall goal rate of execution for the tasks.

We define a minimum dwell time *t_min* as the miniminum dwell time for the smallest task, a task of size *m_min* or less. As discussed above, a task's memory dwell time depends on its size; the larger the task the longer the dwell time. Specifically, the dwell time for a task with memory requirement *m* is *bin(m)* * *t_min*, where *bin(m)* is the smallest power of two greater than or equal to *m / m_min*. Requiring dwell times to step by powers of two allows for buddy-style coalescing of memory-time allocations between small tasks allocations and large task allocations.

The scheduler uses a first-fit strategy for scheduling available memory. The scheduler is invoked whenever a block of memory becomes available. A list of swapped-out but ready-to-run tasks is maintained, sorted by rank, where tasks with a lower rank are considered first. The scheduler selects the first task to be scheduled. If the memory available is greater than the task's memory requirement, the task is scheduled to be swapped in, the available memory is reduced by the size of the task, and the task's rank is updated to reflect its current residency. Otherwise, the next task in the list is examined. This procedure continues until either all the memory is scheduled or the available memory is less than the size of the smallest task.

Thus the scheduler's job can be thought of as packing rectangular boxes along a single dimension *M* wide, where *M* is the amount of memory available. The scheduler sorts the scheduled tasks by dwell time expiration. When the dwell time expires for a set of tasks, a memory quantum has expired. At each quantum expiration, the scheduling algorithm is repeated to allocate the available memory to a (possibly) new set of tasks. The algorithm also ensures that the currently executing tasks are not swapped out if no higher ranking tasks is waiting to be scheduled.

The memory scheduler is notified when an executing task blocks. If the task's memory residence time exceeds a minimum residency requirement, the task is swapped out and placed on a blocked list. When the task is unblocked, it is inserted in the ready-to-run task list.

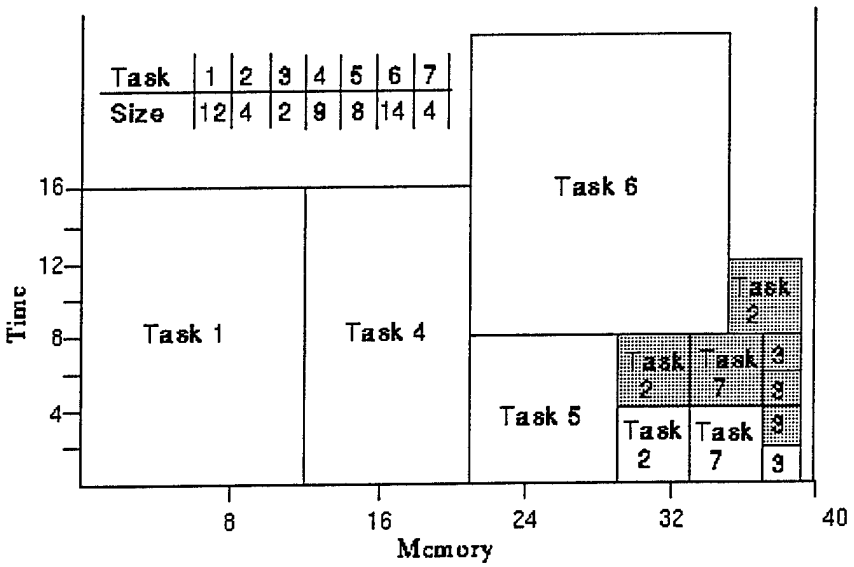

Figure 3. Example Memory Schedule

Figure 3 illustrates an example schedule for 7 tasks. The 7 tasks are listed in rank order by their sizes. The horizontal axis represents memory, and the vertical axis represents time. Each rectangle represents an allocation of memory to a task. The duration of the schedule varies with the dwell times of the tasks scheduled. Note that task 6 cannot be scheduled initially, since its size is greater than the memory available after tasks 1 through 5 are scheduled. Also, even though task 4 requires only slightly more memory than task 5, the dwell time of task 4 is twice the dwell time of task 5, since a task's dwell time is a factor of the smallest power of two greater than its size.

Each horizontal line in Figure 3 represents a quantum expiration time at which a set of tasks becomes eligible to be swapped out of memory. At this time, the scheduling algorithm is repeated. Task 3's quantum expires first. Assuming there are no other tasks in the system, task 3 will be rescheduled (this is represented in the figure by a shaded rectangle). Similarly, tasks 2 and 7 will be rescheduled until time 4. At that time, assuming tasks 3 and 5 drop in rank below task 6, tasks 3, 5 and 7 will be swapped out and task 6 will be scheduled. The next quantum expires at time 9, and the scheduling algorithm is repeated (scheduling decisions at time 9 and beyond are not shown in Figure 3).

When a new task arrives, it is placed in the task list in sorted order using its assigned rank. When a task exits, the scheduler tries to fill the task's memory with another task on the task list if the remaining dwell time is long enough to support the overhead of swapping the new task into memory.

The memory schedulers must also handle requests by the tasks for dynamic memory allocation. A subset of the available memory is reserved to handle these requests. Still, if a task's remaining dwell time is short or if the memory is not available, the task's memory allocation request fails. A message is sent to the task's processor scheduler to stop the task. Once the task is stopped, it is swapped out, and if it had sufficient remaining dwell time, another task from the task list is chosen, if possible, to occupy the vacated memory for the remaining dwell time. The total memory requirements for the task is incremented so that the next time the task is swapped in, it receives the larger allocation. If a small

memory task requests a total allocation in excess of the class of tasks handled by the ML-scheduler, then the task is placed under the control of the MB-scheduler.

When memory becomes available, either through a task's exiting or freeing memory, the MB-scheduler looks at the remaining dwell time of the freed memory. If a large fraction of the dwell time remains, the scheduler looks for an appropriately-sized task in the swapped-out task list.

Starvation Avoidance

The scheduler's first-fit scheduling strategy is a straightforward and simple policy designed to provide high memory utilization. However, this simple strategy alone is not sufficient to avoid starvation, since there is no guarantee that a large enough block of memory will eventually become available to schedule a large memory task.

In order to avoid large task starvation, when the available memory is not sufficient to schedule the next task on the swapped-out task list, the earliest possible time is determined at which enough memory will be available to schedule the task, given the current schedule. The scheduler will refuse to schedule subsequent tasks that do fit in available memory if their dwell time would result in the higher ranking task not being schedulable at the later time. This gives huge jobs a chance to run within a time interval corresponding to their rank.

Extensions

The memory schedulers do not consider a task's processor requirements when scheduling tasks to be swapped in. An extension could consider the number of teams required by the task as well as the task's memory to try to schedule sets of jobs in memory concurrently that don't overutilize or underutilize the processing resources.

Currently, memory is statically divided between the MB-scheduler and the ML-scheduler. To increase the utilization of memory, the MB-scheduler could release unallocated memory to the ML-scheduler for use until the next MB-scheduler quantum expires. Similarly, a more flexible partition could be implemented, where the amount of memory reserved for the small memory tasks varies with the changing load exerted on the system by the batch and interactive tasks. However, research by *Ashok and Zahorjan* suggests that the benefits of such dynamic partitions are not easily attained.

5. Conclusions

This paper discussed several novel characteristics of the Tera memory schedulers. Jobs are distinguished by rank, which is a function of its demand for memory and its current allotment. We described an algorithm for swapping jobs that minimizes overhead and showed that swapping overhead grows as the sum of the squares of the sizes of jobs being swapped. Perhaps surprisingly, the overhead is independent of the order of swapping; this means we can alter the order, should the need arise, without incurring additional overhead or computational effort.

6. References

R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith.

The Tera computer system.
In *1990 International Conference on Supercomputing*, June 1990.

I. Ashok and J. Zahorjan.
Scheduling a mixed interactive and batch workload on a parallel, shared memory supercomputer.
In *Supercomputing 1992*, Nov 1992.

D. Burger, R. Hyder, B. Miller, and D. Wood.
Paging tradeoffs in distributed-shared-memory multiprocessors.
In *Supercomputing '94*, November 1994.

Raymond Essick.
An event-based fair share scheduler.
In *Winter 90 USENIX Conference*, January 1990.

Raphael Finkel.
An Operating Systems *Vade Mecum.*
Prentice-Hall, 1986, p. 24.

G. J. Henry.
The fair share scheduler.
Bell Laboratories Technical Journal, 63(8), October 1984.

For more information, contact Tera Computer Company <info@tera.com>

IEEE Transactions on Parallel and Distributed Systems, April 1991.

J. Zahorjan and C. McCann.
Processor scheduling in shared memory multiprocessors.
In *ACM SIGMETRICS Conference*, May 1990.

For more information, contact Tera Computer Company <info@tera.com>
